GitHub Gist

bawjensen / gist:7819463
Created just now

gistfile1.cpp C++

```cpp
/*
Jensen_A6.cpp
12/06/13
Bryan Jensen

This program contains the declarations and functionality for a hash table storing strings, counts
(for number of times inserted) and a counter for the number of collisions at each slot. This hash table
implements closed hashing with quadratic probing, and is used to analyze the quality of hashing achieved
using Knuth's constant of 0.618 versus other values.

Description: This file contains prototypes as well as functionality for the entirety of two classes, one
                         for the hash table as a whole, the other for the slots in the table, each storing the string,
                         count and number of collisions. It also contains the main() "driver" for these classes.
*/

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string.h>
#include <string>

using namespace std;

/******************************* Slot Class for HashTable  ******************************/
/*************************** Class Definition and Prototypes ***************************/

class HashTableSlot {
private:
protected:
        string data;                       // Data contents of the slot in the hash table
        long count;                        // Count of the number of times that data item has been inserted
        long numCollisions;                // Number of collisions that have occured at this slot
public:
        HashTableSlot() { this->data = ""; this->count = 0; this->numCollisions = 0; }; // Default constructor
        HashTableSlot(string, long);       // Parameterized constructor

        string getData() { return this->data; }; // Getter for data attribute
        long getCount() { return this->count; }; // Getter for count attribute
        long getNumCollisions() { return this->numCollisions; }; // Getter for numCollisions attribute

        bool empty() { return this->data == ""; }; // Returns if the table slot is empty (the string data is the empty string)
        void incrementCount() { this->count++; }; // Increments the count of the number of times that this string has been inserte
        void incrementCollisions() { this->numCollisions++; }; // Increments the count of the number of collisions at this table
};

/********************************** Method Definitions *********************************/
/************************************ Helper Methods ************************************/

// Inserts the key into the hash table slot, with the initial count of keyCount. Initializes
// numCollisions at 0.
// Precondition: key is a valid, non-empty string to insert into the hash table
// Postcondition: A HashTableSlot object has been created with the key, keyCount and 0 values.
HashTableSlot::HashTableSlot(string key, long keyCount) {
        this->data = key;
        this->count = keyCount;
        this->numCollisions = 0;
}

/********************************** Hash Table Class ************************************/
/*************************** Class Definition and Prototypes ***************************/

class HashTable {
private:
protected:
        HashTableSlot* table;              // Array of slots in the table
        HashTableSlot* table;              // Array of slots in the table
        float hashConstant;                // Constant value for hashing function
        long tableSize;                    // Value for the length of the closed hash table
        long displayLength;                // Minimum number of items to display from table

        long hash(string);                 // Method to hash a string to an int for storage location
        bool full();                       // Method to check if the hash table is full or not (for error checking)
public:
        HashTable() : table(NULL), hashConstant(0), tableSize(0), displayLength(0) {}; // Default constructor

        // Setters
        void setTableSize(short);          // Setter for the tableSize attribute
        void setHashConstant(float);       // Setter for the hashConstant attribute
        void setDisplayLength(short);      // Setter for the displayLength attribute

        // Public functionality methods
        void insert(string);               // Inserts a string into the table, or increments the count of
                                           // the slot containing the string if already present
        void display();                    // Setter for the tableSize attribute
};

/********************************** Method Definitions **********************************/
/************************************ Helper Methods ************************************/

// Converts a string value into an int representing the intended insertion location of the
// string key.
// Precondition: None
// Postcondition: An int is returned to the caller representing the insertion location of the key into the hash table
long HashTable::hash(string keyString) {
        long keyValue = 0;

        for (long i = 0; i < strlen(keyString.c_str()); i++) {
                keyValue += keyString[i];
        }

        float hashValue = keyValue * this->hashConstant;

        hashValue -= (long)hashValue; // Get only the decimal portion of the number
        hashValue *= this->tableSize;

        return (long)hashValue;
}

// Returns a boolean representing whether or not the hash table is full
// Precondition: the table attribute has been initialized with an array of tableSize length
// Postcondition: Returns to the caller whether the table is full
bool HashTable::full() {
        for (long i = 0; i < this->tableSize; i++) {
                if (this->table[i].empty()) {
                        return false;
                }
        }
        return true;
}

/************************************ Setter Methods ************************************/

// A setter for the tableSize variable. NOTE: also resets the contents of the table, as opposed
// to copying them over to the new table.
// Precondition: None
// Postcondition: The hash table contains a new (empty) table of size tSize
void HashTable::setTableSize(short tSize) {
        this->tableSize = tSize;
        if (this->table) delete[] this->table;
        this->table = new HashTableSlot[tSize];
}

// A setter for the hashConstant attribute.
// Precondition: None
// Postcondition: The hashConstant attribute has been updated with the given value
void HashTable::setHashConstant(float hConstant) {
        this->hashConstant = hConstant;
}

// A setter for the displayLength attribute.
// Precondition: None
// Postcondition: The displayLength attribute has been updated with the given value
void HashTable::setDisplayLength(short dLength) {
        this->displayLength = dLength;
}

/*********************************** Public Methods ***********************************/

// Inserts an item into the hash table, using the hash method and quadratic probing to solve
// insert conflicts.
// Precondition: The table has enough free slots to accommodate the new insertion. The program exits if not so.
// Postcondition: The key string has been inserted into the hash table at the hash value, shifted over as necessary
//                                by the quadratic probing.
void HashTable::insert(string key) {
        if (this->full()) {
                cout << "Hash table is full. Use larger table size in future." << endl;
                exit(1);
        }

        long keyIndex = hash(key);
        long quadraticProber = 1;
        long probeIndex = keyIndex; // Initialize probing at index

        while ( !( this->table[probeIndex].empty() or this->table[probeIndex].getData() == key ) ) {
                // cout << key << " hashed or probed to the same slot as " << this->table[probeIndex].getData() << endl;
                this->table[probeIndex].incrementCollisions();

                probeIndex = ( probeIndex + (quadraticProber * quadraticProber) ) % this->tableSize;
                // probeIndex = ( keyIndex + (quadraticProber * quadraticProber) ) % this->tableSize; // Alternative method of pro
                quadraticProber++;
        }

        if (this->table[probeIndex].getData() == key) {
                this->table[probeIndex].incrementCount();
        }
        else {
                this->table[probeIndex] = HashTableSlot(key, 1);
        }
}

// Displays the table to stdio in a formatted output.
// Precondition: None
// Postcondition: The contents of the table have been output to stdio (the console) in
//                                a formatted fashion.
void HashTable::display() {
        short colWidths[] = { 5, 20, 10, 15 };
        // Header row formatting
        cout << setw(colWidths[0]) << "Index"
                 << setw(colWidths[1]) << "Word"
                 << setw(colWidths[2]) << "Count"
                 << setw(colWidths[3]) << "Collisions" << endl;

        // Creates dynamically width bar from string constructor which uses a number and char to construct a monotone string
        cout << string(colWidths[0] + colWidths[1] + colWidths[2] + colWidths[3], '-') << endl;

        long i = 0;
        long numShown = 0;
        while (i < this->tableSize and numShown < this->displayLength) {
                if (!this->table[i].empty()) {
                        cout << setw(colWidths[0]) << i
                                 << setw(colWidths[1]) << table[i].getData()
                                 << setw(colWidths[2]) << table[i].getCount()
                                 << setw(colWidths[3]) << table[i].getNumCollisions() << endl;
                        numShown++;
                }

                i++;
        }

        if (numShown < this->displayLength) {
                cout << "Note: attempting to display more elements than are in table." << endl;
        }

        int numTotalCollisions = 0;
        for (int i = 0; i < this->tableSize; i++) {
                numTotalCollisions += this->table[i].getNumCollisions();
        }
        cout << "Table had " << numTotalCollisions << " collisions in total." << endl;
}

/********************************** Helpful functions **********************************/
```

```cpp
/*********************************** Helpful functions **********************************/

// Uses user input to create a hash table.
// Precondition: None
// Postcondition: A hash table has been created from the values input by the user
//                                  and returned to the caller.
HashTable userCreateHash() {
        short tableSize, displayLength;
        float hashConstant;
        string filename;
        HashTable newHash;

        cout << "Enter table size: ";
        cin >> tableSize;
        cout << "Enter value for A (the hash constant): ";
        cin >> hashConstant;
        cout << "Enter file name: ";
        cin >> filename;
        cout << "Number items to display: ";
        cin >> displayLength;

        // tableSize = 201;
        // hashConstant = .618;
        // filename = "temp";
        // displayLength = 201;

        newHash.setTableSize(tableSize);
        newHash.setHashConstant(hashConstant);
        newHash.setDisplayLength(displayLength);

        ifstream inFile;
        inFile.open(filename.c_str());

        if (inFile.fail()) {
                cout << "Whoops, bad filename." << endl << "Exiting." << endl << endl;
                exit(1);
        }

        string fileInput;
        inFile >> fileInput;

        while (inFile) {
                newHash.insert(fileInput);
                inFile >> fileInput;
        }

        return newHash;
}

/***************************************** Main *****************************************/

int main() {
        HashTable myHash = userCreateHash();
        myHash.display();

        return 0;
}
```